AD-F951 302

(2)

AD-A203 930

TECHNICAL REPORT RD-GC-88-38

A DESCRIPTION OF THE COMPUTER SIMULATION OF A NEW BUS ARBITRATION SCHEME

G. Patton Bradford
Guidance & Control Directorate
Research, Development, & Engineering Center

DECEMBER 1988

U.S. ARMY MISSILE COMMAND
Redstone Arsenal, Alabama 35898-5000

Approved for public release; distribution unlimited.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188
Exp. Date: Jun 30, 1986

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) RD-GC-88-38	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Guidance & Control Directorate RD&E Center 6b. OFFICE SYMBOL (If applicable) AMSMI-RD-GC-S	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Commander, US Army Missile Command ATTN: AMSMI-RD-GC-S Redstone Arsenal, AL 35898-5254	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Same 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

A DESCRIPTION OF THE COMPUTER SIMULATION OF A NEW BUS ARBITRATION SCHEME (U)

12. PERSONAL AUTHOR(S)
G. Patton Bradford

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final Technical	FROM Mar 88 TO May 88	DECEMBER 1988	30

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Computer bus, computer architecture, arbitration

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The object of bus arbitration schemes is to provide fair access to the bus by the various elements of the computer system. Arbitration must allow for each element to have access to the bus while, at the same time, preventing any single element from monopolizing the bus. The method of bus arbitration presented here appears to meet these goals.

The arbitration scheme described in this report places the various elements of the system on a rotating system of priorities. After the various priorities are set initially (0...N-1, where N is the number of system elements) all the priorities are incremented by one on a modulo N basis. In this way, each processor will eventually have top priority and be able to take possession of the bus if needed.

In a simulation of this scheme using a random mix of bus demands, the average wait of any processor for the bus was very short, generally less than 10 bus cycles ("bus cycle" defined in the body of the report.) For simulations in which the bus demand was high (a probability

(cont'd)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL G. Patton Bradford	22b. TELEPHONE (Include Area Code) (205)876-7081 22c. OFFICE SYMBOL AMSMI-RD-GC-S

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

DD 1473 (cont'd)

Item 19. Abstract

of requesting the bus of 0.75 and a probability of retaining the bus of 0.75), the average delay was seven cycles.

The simulation program itself is described in detail and the results of various combinations of probabilities are presented.

ACKNOWLEDGEMENTS

I must be honest and state right from the beginning that this arbitration scheme is not mine; I did not think of it, and I can in no way lay claim to it. This scheme is the idea of Mr. James Holeman of Systems Dynamics, Inc. During some discussions on the design of an experimental digital signal processing board, Jim presented this idea as a simple, reliable, and robust method for arbitrating access to a common bus between various system boards. This simulation appears to bear out some of his hopes for the design in that it provides for short average delays as well as a relative immunity to system failure due to the failure of system components.

Thanks, Jim. I appreciate it.

TABLE OF CONTENTS

I. INTRODUCTION

One problem that continues to face the designer of computer systems in which various system elements must share common resources is the arbitration of access to these resources. In most systems, this comes down to the sharing of a common bus that connects these resources to each other. The question is: how can access to this resource, the bus, be handled in a "fair" manner? That is, how can one arrange for every potential user of the bus to have adequate access to the bus while preventing any one user from monopolizing it? Also, how can this be done in a way that is the least taxing in terms of hardware and software?

The method put forth in this paper for handling this problem is a conceptually simple solution involving a rotating priority system. In this way, all boards at some point would have exclusive, and in some cases non-interruptible, access to the bus. Because the priority rotates, no one individual user of the bus can monopolize it. This scheme is described in the next section.

Following this discussion of the arbitration algorithm is an extensive treatment of the program used to simulate the action of the arbitration scheme. Finally, there is a discussion of the results of a number of runs using different values for the probabilities of requesting the bus and of completing a bus access.

II. ARBITRATION ALGORITHM DESCRIPTION

The arbitration scheme itself is conceptually very simple. The N processors, or bus users, are initially assigned priorities of 0 to N-1, each processor receiving a different priority. While no accesses are made to the bus, or, more accurately, no processor releases the bus, the priorities remain the same. However, when a processor obtains and then releases the bus, all processors have their priorities incremented by one on a modulo N basis; that is, the processor with priority 0 goes to priority 1, priority N-2 goes to N-1, and priority N-1 goes to 0. In this way, the highest priority processor, the one with the greatest probability of having had access to the bus now receives the lowest priority. A "fairer" method would have the processor releasing the bus receive the lowest priority; however, this would require some method for doing a wholesale reassignment of priorities, whereas the method described requires only incrementing counters on each processor.

However, once a processor receives the bus, it does not necessarily have non-interruptible access to it; the processor may lose the bus to a processor having higher priority. If this happens, the executing process is blocked, the processor relinquishes the bus, the priorities of all the processors are updated, and the requesting processor with the highest priority is granted the bus. The blocked processor now has a pending bus request that will be answered when its priority again exceeds that of all other requesting processors.

III. BUS SIMULATION PROGRAM

In order to develop an understanding as to how this sytem would perfor-over time, a program was written that will simulate the performance of this type of scheme over a number of iterations. To allow for a degree of variability, different parameters were set up in the form of probabilities: in particular, the probability of a processor failing, of requesting the bus, and of the processor completing its bus access at any particular time. In this manner, different bus loads could be simulated. The basic structure of the simulation is described below. (See Appendix A for the source code listing.)

The processors are represented in the program as a relatively large data structure containing various flags, probabilities, counters, and value-holders. The flags for each process are: ACTIVE, which shows if the processor is healthy; BLOCKED, which is set if the processor has the bus but has been interrupted; WAITING, which is set if the processor has a bus request which has not been met; and RUNNING, which indicates that the processor has run or is running since the last time update. The probabilities contained in each process structure are those of failure (PROB_FAIL), making a bus request (PROB_REQUEST), and of completing a bus request on any given cycle (PROB_COMPLETE). The counters contain information on the cycles used in the present block, wait, or bus access (PRESENT_BLOCK, PRESENT_WAIT, and PRESENT_RUN), the total number of cycles spent in blocks, waits, or bus accesses (TOTAL_BLOCK, TOTAL_WAIT, NUM_BLOCKS, NUM_WAITS, and NUM_RUNS). The longest of any individual block, wait, or run is also tracked (LONGEST_BLOCK, LONGEST_WAIT, LONGEST_RUN). Finally, and perhaps most important to this simulation, is PRIORITY, which contains the processor's present priority.

To begin the simulation each processor structure has its values initialized through the procedure INIT(). The ACTIVE flag is set, the BLOCKED, WAITING, and RUNNING flags are cleared, the counters are cleared, and using the procedure SET_PRIORITY(), the priority of each processor is randomly set to an integer value between 0 and N-1 (since in this simulation the processors cannot run or request the bus concurrently, they must instead be polled sequentially.) This random setting of the priorities of the different processors helps to nullify any effects that might occur due to having the processors arranged sequentially by priority. After this, the various probabilities are set. The program allows for this to be done randomly with a default value or with the user specifying the probability for each processor individually. These options allow for the tailoring of the bus loading to one's preference.

The main program itself consists mainly of two large loops. The outer loop determines how many times polling of the processors occurs and the inner loop determines how each processor will act when it is polled. At the begining of the outer loop are print statements that report the status of the system every thousand iterations. At the end of the outer loop is a call to the UPDATE_TIME() procedure which updates the counters within each processor structure, incrementing as necessary the number of cycles in the RUN, WAIT, or BLOCK states. The inner loop is where the actual computations that determine processor state are performed. The routine will check the referenced processor's current state and then, based on this state and the probabilities associated with this processor, the next state of the processor is determined.

At the beginning of this inner loop, the processor under consideration may be in one of two main states, either healthy and functioning (ACTIVE=1) or "dead" (ACTIVE=0), having failed at some point in the past. If the processor is dead, then the program continues on to the next processor. Figures 1 through 3 are flow charts of this portion of the program.

On the other hand, if the processor has not failed, it will be in one of four states as it enters processing. It will either be the present bus master, it will be blocked (it had the bus at one time but was preempted before it could complete the transaction), it will be waiting (the processor has requested the bus, but has not yet received it), or it could have no pending requests. Each of these states requires that different actions be taken and each have different alternatives to choose. Each of these alternatives is chosen through the use of the probabilities assigned to each processor and the actions of a random number generator. Any time the value received from the random number generator is less than the value of the assigned probability, the action associated with that probability will be performed.

The first state to be looked at is that of the processor being the present bus master. This is perhaps the simplest of all the states to process. The first check is to see if the processor fails at this point. If it does, then its ACTIVE flag is cleared, BUS_MASTER and PRIORITY are set to FREE, and UPDATE_PRIORITY() is called. If the processor remains active, then RUN_PROCESS() is called to see if the processor will complete its access this cycle. The program then continues on to the next processor.

RUN_PROCESS() is a very simple routine itself. If the PROB_COMPLETE associated with this process is greater than some random number, then the bus is freed again. UPDATE_PRIORITIES() is then called.

With the next three states, BLOCKED, WAITING, and NO_PENDING_REQUESTS, again the possibility of processor failure is checked first, and, with these three, the processing is the same. If the processor does fail, ACTIVE is cleared, LONGEST_WAIT, LONGEST_BLOCK, TOTAL_WAIT, and TOTAL_BLOCK are all updated with respect to PRESENT_WAIT and PRESENT_BLOCK. The flags WAITING and BLOCKED are then cleared.

The next check made on processors in one of these three states is whether or not it will make a request for the bus. For the BLOCKED and WAITING states this is automatic. For the NO PENDING_REQUESTS state, this is checked through its PROB_REQUEST value. If there is no request, the program goes on to the next processor.

Next, having made a request for the bus, the priority of the requester is compared to that of the present bus master, that is, to the value stored in the global variable PRIORITY. If the priority of the requester is the lesser of the two values, then the program continues on with the next processor; otherwise, the requesting processor becomes the new bus master. If the bus is free, then this is accomplished by writing the processor's number into BUS_MASTER, its present priority into PRIORITY, and its BLOCKED and WAITING flags cleared. After this, RUN_PROCESS() is called to see if it will complete its access in this cycle. The program then continues on to the next processor.

If, on the other hand, the bus is not free when the requester receives the bus, the old bus master must be blocked. This is accomplished by setting the bus master's BLOCKED flag, incrementing its NUM_BLOCKS, and calling UPDATE_PRIORITIES(). The requester is then given control of the bus as in the previous paragraph.

After the program has stepped through all N of the processors, it is considered to have completed one BUS CYCLE. At the end of each bus cycle, the routine UPDATE_TIME() is called in order to keep track of the amount of time spent by each processor, either on the bus, waiting for initial access to the bus, or in a blocked state. The routine looks at each processor data structure, looking at what flags are set, and updating the appropriate times. First, it looks at the RUNNING flag. If it is set and the processor is the present bus master, then PRESENT_RUN is incremented and LONGEST_RUN is updated. If it is not the bus master, then PRESENT_RUN is cleared. In any case, the RUNNING flag is cleared. Next, UPDATE_TIME() checks the BLOCKED and WAITING flags and updates either PRESENT_WAIT or PRESENT_BLOCK, respectively. Finally, the routine sets the RUNNING flag of the PRESENT_BUS master.

At this point, there may exist some confusion over why UPDATE_TIME() clears and then sets the RUNNING flag while the other flags are left alone. The reason is that, with the structure of the simulation, once a processor is put on WAIT or BLOCK, it remains there until the next bus cycle. On the other hand, if it is running, it may be blocked any time during the present cycle or some future cycle. In order to keep track of the amount of time some processor had the bus, it is assumed that the processor keeps the bus for one entire cycle if it had the bus for any portion of that cycle. For this reason, when a processor is blocked, the RUNNING flag is not reset until UPDATE_TIME() is called at the end of the cycle. In this way, all the processors that ran during the cycle get credit for that cycle, and by setting the bus master's RUNNING flag again, it will get credit the next time the update routine is called at the end of the next cycle.

After the outer loop has run the required number of iterations (10,000 in this simulation), the results are tallied. The total amount of time spent running in a WAIT or in a BLOCK is tallied, as well as the total number of times processors were put in a RUN, WAIT, or BLOCK. From these, average times may be computed. Also reported are the longest times in any state.

IV. SIMULATION RESULTS

Appendix B gives the results of a number of runs. The first three use random values for the probabilities of failure, request, and completion. (The intermediate results from the first run are presented in order to give the reader an idea of what is displayed during a run.) As can be seen from these, although the total times and longest times spend in a WAIT or BLOCK state have considerable variation from one run to another, the average time in a WAIT or BLOCK remains consistently small, generally less than 10 cycles. In addition, the total number of cycles in which the bus was active remained relatively constant.

The rest of the runs presented represent situations that range from a very heavily loaded bus (a high probability of requesting the bus along with a low probability of completing an access) to a very lightly loaded bus (a low probability of request with a high probability of completion.) As can be seen on the heavy loading run, although the number of cycles the bus is is use increases by about half over the random loading, the amount of time in a BLOCK or WAIT more than doubles. However, the average time spent in a BLOCK or WAIT is still under 10 cycles - the same as with the random case. The next case, high probability for both request and completion, is perhaps the ideal bus loading situation. Bus utilization is more than doubled over any other case. In addition, the average WAIT and BLOCK is down to one.

The case in which both the probabilities are low (a request is not likely, but if there is one, it is not likely to complete on any given cycle) is about average when compared to the other cases. The average delays are in the same range as all the others, under 10 cycles, and because there are fewer requests, the bus is used less often than the higher request situations, even the one with the high blocking rate. The final situation, with low request probability and high completion probability, has the lowest rate of blocking and waiting, though not much lower than the high request/high complete. Again, because the request rate is low, bus utilization is dropped.

V. CONCLUSIONS

This system of bus arbitartion using a rotating priority scheme appears to be an effective and efficient method of handling bus arbitration. Average delays are consistently low in all cases tested; failure of individual system elements appears to have little effect on overall system performance; and the conceptual simplicity of the system should allow for relatively efficient implementations in hardware.

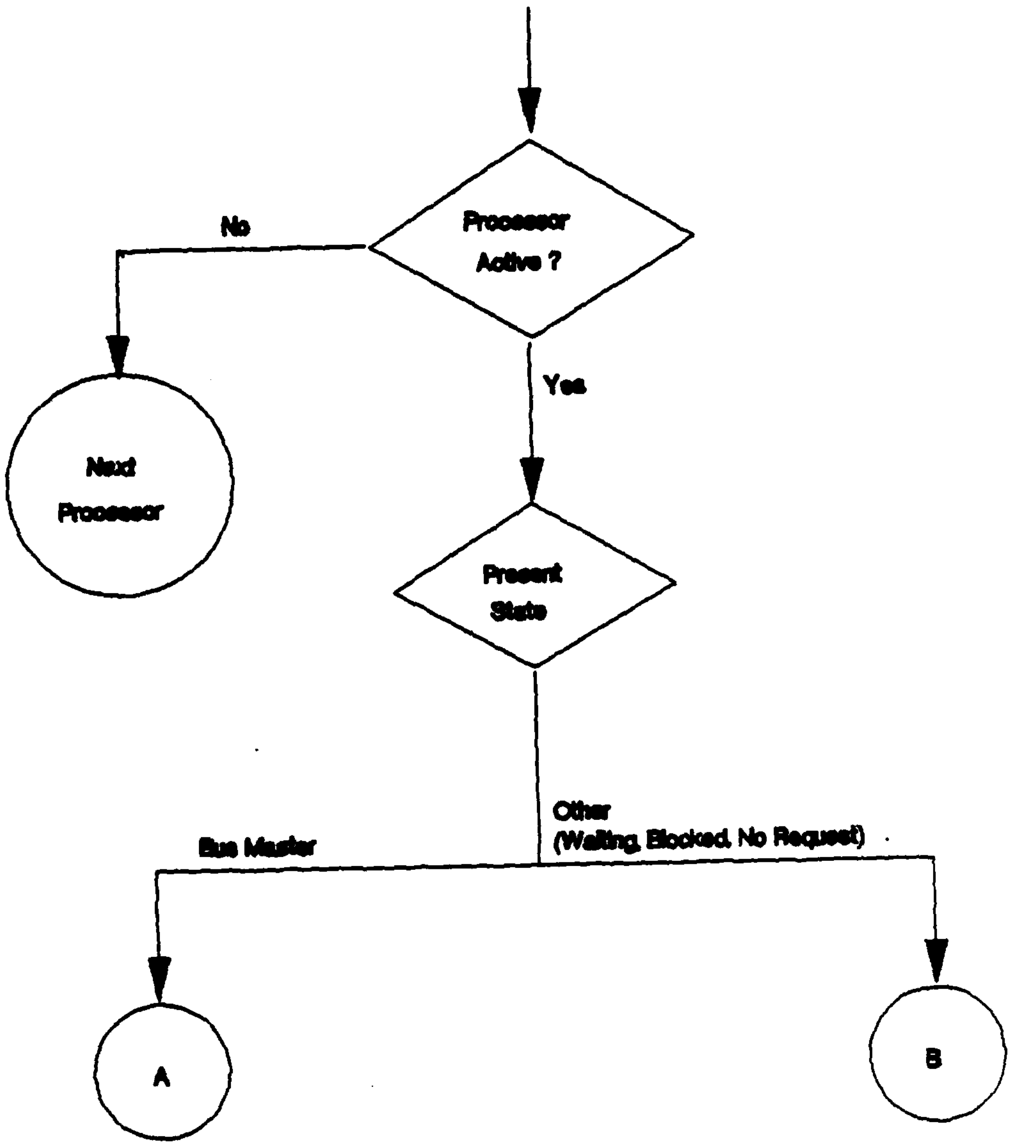

Figure 1. Flowchart for inner processing loop.

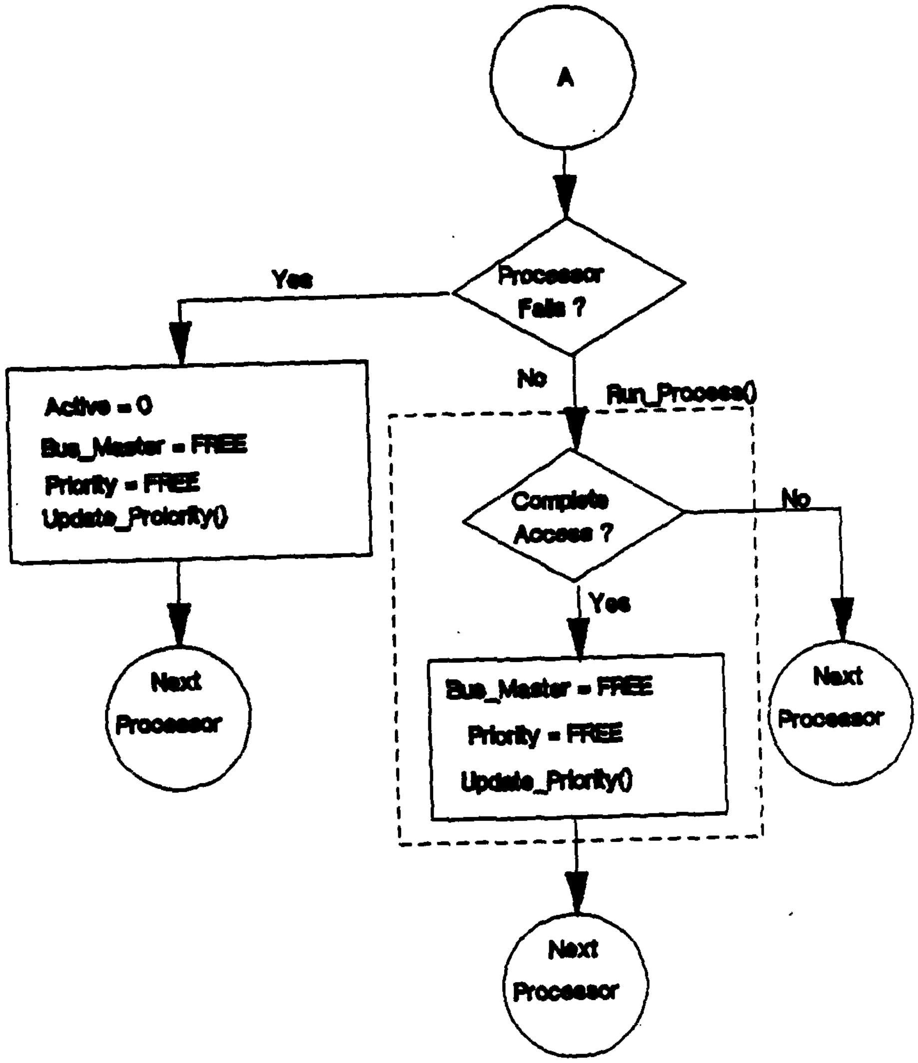

Figure 2. Flowchart of bus master processing loop.

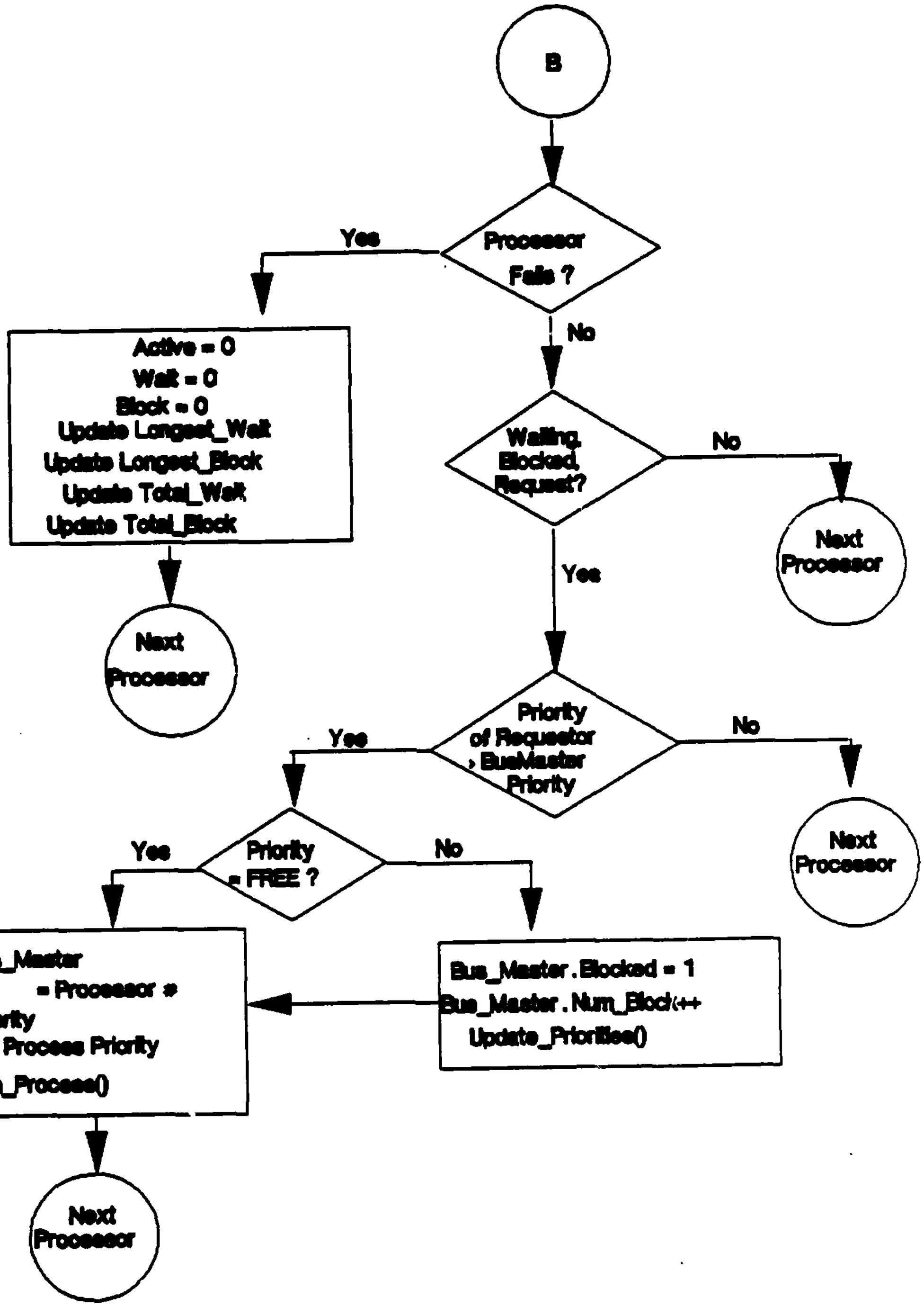

Figure 3. Flowchart of other (waiting, blocked, no request) processing loop.

APPENDIX A

SOURCE CODE LISTING

```
/**************************************************************************
/*
/*    bussim.c - Program to simulate the rotating priority bus arbitration
/*    scheme.  The program will run the main cycle (all ten processors being
/*    polled once) 10,000 times.  It will then report on the maximum and
/*        average delays encountered.
/*
/*************************************************************************/

/* #include "include.h" */

#include <stdio.h>
#include <dos.h>

/**************************************************************************
/*
/*    global.h  - Defines such things as the structure for the processes
/*
/*************************************************************************/

#define NUMPROCESSORS 10
#define FREE -1
#define MAXLOOPS 10000

struct Process
{
     char Active; /* Flag showing whether or not the processor */
                  /* has failed */
     char Blocked;    /* Had been processing but was interrupted */
     char Waiting;        /* Has requested bus but has not received it */
     char Running;        /* Has run during this cycle */

     short int Priority;  /* Present priority of processor */

     float ProbFail;  /* The probability that this processor will fail */
     float ProbRequest;   /* The probability that a bus request will be */
                /* be made on some cycle */
     float ProbComplete;  /* The probability that, if the processor has */
                /* the bus, it will complete this cycle */

     int PresentWait; /* The number of cycles waiting on present request */
     int LongestWait; /* The longest wait on a bus request */
     int TotalWait;       /* Total number of cycles in a wait state */
     int NumWaits;        /* The number of times processor made to wait */
     int PresentBlock;    /* The number of cycles blocked */
     int LongestBlock;    /* The longest time blocked */
     int TotalBlock;  /* Total number of cycles in a blocked state */
     int NumBlocks;       /* The number of times executing process is
                      blocked */
     int PresentRun;  /* The number of non-interrupted cycles possessing
                the bus */
     int LongestRun;  /* The longest time spent in any single run */
```



```
     int TotalRun;           /* The total amount of time this processor has
                    had the bus */
     int NumRuns;           /* The number of times the processor has requested
                    and received the bus */

}  ProcArray[NUMPROCESSORS];

long  Seed;        /* The seed value for the random number generator */
short  BusMaster = FREE;  /* Present owner of the bus */
short  Priority = FREE;   /* Priority of owner of the bus */
long SumWaitTime = 0L;  /* Sum of TotalWait */
long SumBlockTime = 0L;   /* Sum of TotalBlock */
long SumRunTime = 0L; /* Sum of TotalRun */
long SumNumWaits = 0L;  /* sum of numwaits */
long SumNumBlocks = 0L; /* sum of numblocks */
long SumNumRuns = 0L;  /* sum of NumRuns */
int MaxWaitTime = 0;  /* longest of LongestWait */
int MaxBlockTime = 0;  /* longest of LongestBlock */
int MaxRunTime = 0;   /* Longest of LongestRun */
int MaxNumWaits = 0;  /* largest number of waits */
int MaxNumBlocks = 0; /* largest number of blocks */
int MaxNumRuns = 0;   /* Largest number of runs */
int AvgWait[NUMPROCESSORS];   /* average time in a wait state */
int AvgBlock[NUMPROCESSORS];  /* average time in a block state */
int AvgRun[NUMPROCESSORS];    /* average time in a bus possession */
int SumAvgWait = 0;
int SumAvgBlock = 0;
int SumAvgRun = 0;

/* End of global.h */

/**************************************************************************
/*
/*    Random() - function to produce a Random number x, 0 <= x < 1.
/*
/*************************************************************************/
```

```
float Random()
/* Routine to produce the random numbers */
{
     Seed = (25173L * Seed + 13849L) % 65536;
     return( ((float)Seed)/((float)65536));

} /* end Random() */


/***********************************************************************
/*
/*   UpdateTime() - Update the wait times on all the processors
/*
/***********************************************************************/
```

PAGE 3
05-03-88
14:22:08

Line# Source Line   Microsoft C Compiler Version 4.00

```
void UpdateTime()
{
     short int i;

     for (i=0; i<NUMPROCESSORS; i++)
     {
          /* Update the time on any processor that has run this cycle */
          if (ProcArray[i].Running)
          {
               if (BusMaster == i)
               {
                    ProcArray[i].PresentRun++;
                    if (ProcArray[i].PresentRun > ProcArray[i].LongestRun)
                         ProcArray[i].LongestRun = ProcArray[i].PresentRun;
               }
               else
                    /* Clear PresentRun if processor does not presently
                       have the bus */
                    ProcArray[i].PresentRun = 0;

               /* Update longestrun and totalrun for all the processors.
                  Clear all of the Running flags. */
               if (!ProcArray[i].LongestRun)
                    ProcArray[i].LongestRun = 1;
               ProcArray[i].TotalRun++;
               ProcArray[i].Running = 0;
          }

          /* Update the time on each blocked or waiting process */
          if (ProcArray[i].Blocked)
               ProcArray[i].PresentBlock++;
          else
          {
               if (ProcArray[i].Waiting)
                    ProcArray[i].PresentWait++;
          }
     }
     if (BusMaster != FREE)
          ProcArray[BusMaster].Running = 1;
} /* End of UpdateTime() */
```

UpdateTime Local Symbols

Name	Class	Offset	Register
i	auto	-0002	

```
/***********************************************************************
/*
/*   SetPriority() - Do random initialization of priorities
/*
/***********************************************************************/
```

PAGE 4
05-03-88
14:22:08

Line# Source Line   Microsoft C Compiler Version 4.00

```
void SetPriority()
{
     char IsSet[NUMPROCESSORS];
     int i, x;

     /* Initialize IsSet to 0 */
     for (i=0; i<NUMPROCESSORS; i++)
          IsSet[i] = 0;

     /* Initialize priorities.  Get random number between 0 and
        NUMPROCESSORS.  If already in use, choose another.  Otherwise,
        assign x to ProcArray[i].Priority and set IsSet[x] */

     for (i=0; i<NUMPROCESSORS; i++)
     {
          do
               x = (int)(10 * Random());
          while (IsSet[x]);
          IsSet[x] = 1;
          ProcArray[i].Priority = x;
     }
} /* End of SetPriority() */
```

SetPriority Local Symbols

Name	Class	Offset	Register
x	auto	-000e	
... .	auto	-000c	
isdet.	auto	-000a	

```
/************************************************************************
/*
/*    UpdatePriority() - Update the priorities on all the processors
/*
/************************************************************************/
void UpdatePriority()
{
     short int i;
     for (i=0; i<NUMPROCESSORS; i++)
     {
          /* To update the priority of each processor, increment each and
             mod by NUMPROCESSORS */
          ProcArray[i].Priority = (++ProcArray[i].Priority) % NUMPROCESSORS;
     }
} /* End of UpdatePriority() */
```

UpdatePriority Local Symbols

Line# Source Line Microsoft C Compiler Version 4.00

Name	Class	Offset	Register
i	auto	-0002	

```
/************************************************************************
/*
/*   RunProcess() - Routine to see if bus access will complete this cycle
/*
/************************************************************************/
void RunProcess(ProbComplete)
float ProbComplete;
{
     if (Random() <= ProbComplete)
     /* Bus Master completes access.  Release the bus and
        update the priorities */
     {
          BusMaster = FREE;
          Priority = FREE;
          UpdatePriority();
     } /* end of Bus Master bus access */
} /* End of RunProcess */
```

RunProcess Local Symbols

Name	Class	Offset	Register
ProbComplete	param	0004	

```
/************************************************************************
/*
/*   init.c  -  Routine to initialize variables for computation
/*
/************************************************************************/
init()
{
     char Answer;
     float Value;
     union REGS inregs, outregs;
     register int i;

     /* Seed random number generator */
     inregs.h.ah = 0x2c;  /* DOS call for Get Time */
     intdos(&inregs, &outregs);
     Seed = (long int)outregs.h.dl;

     /* Set the initial priorities for all of the processors */
     SetPriority();
```


Line# Source Line Microsoft C Compiler Version 4.00

```
     for (i=0; i< NUMPROCESSORS; i++)
     {
          ProcArray[i].Active = 1;    /* Processor is functional*/
          ProcArray[i].Blocked = 0;
          ProcArray[i].Waiting = 0;
          ProcArray[i].Running = 0;
          ProcArray[i].PresentWait = 0;
          ProcArray[i].LongestWait = 0;
          ProcArray[i].TotalWait = 0;
```

```
            ProcArray[i].NumWaits = 0;
            ProcArray[i].PresentBlock = 0;
            ProcArray[i].LongestBlock = 0;
            ProcArray[i].TotalBlock = 0;
            ProcArray[i].NumBlocks = 0;
        }/* Set the values for the various probabilities */

        /* ProbFail */
pick1:      printf("Do you wish to set the probabilities of failure\n");
        printf("(R)andomly\n(D)efault value, or\n(I)ndividually?\n");
        scanf(" %c", &Answer);

        switch ((int)Answer)
        {

            case 'R':
            case 'r':
                for(i=0; i<NUMPROCESSORS; i++)
                {
                    ProcArray[i].ProbFail = .0001 * Random();
                    /* Low probability is required to keep them from
                       all failing during the first few iterations.
                    */

                    printf("Probability of failure %d:  %f\n",i,ProcArray[i].ProbFail);
                }
                break;
            case 'D':
            case 'd':
                printf("Default value?\n");
                scanf("%f", &Value);
                for (i = 0; i < NUMPROCESSORS; i++)
                    ProcArray[i].ProbFail = Value;
                break;
            case 'I':
            case 'i':
                for (i = 0; i < NUMPROCESSORS; i++)
                {
                    printf("\nProbability of failure for processor %d:  ",i);
                    scanf("%f", &Value);
                    ProcArray[i].ProbFail = Value;
                }
                printf("\n");
                break;
```



```
            default :
                printf("Dumb.  Very dumb. Try again.\n");
                goto pick1;
        }
        printf("\n");
        /* ProbRequest */
pick2:      printf("Do you wish to set the probabilities of making a bus request\n");
        printf("(R)andomly\n(D)efault value, or\n(I)ndividually?\n");
        scanf(" %c", &Answer);

        switch ((int)Answer)
        {

            case 'R':
            case 'r':
                for(i=0; i<NUMPROCESSORS; i++)
                {
                    ProcArray[i].ProbRequest = Random();
                    printf("Probability of Bus Request %d:  %f\n",i,ProcArray[i].ProbRequest);
                }
                break;
            case 'D':
            case 'd':
                printf("Default value?\n");
                scanf("%f", &Value);
                for (i = 0; i < NUMPROCESSORS; i++)
                    ProcArray[i].ProbRequest = Value;
                break;
            case 'I':
            case 'i':
                for (i = 0; i < NUMPROCESSORS; i++)
                {
                    printf("\nProbability of request for processor %d  ",i);
                    scanf("%f", &Value);
                    ProcArray[i].ProbRequest = Value;
                }
                printf("\n");
                break;
            default :
                printf("Dumb.  Very dumb. Try again.\n");
                goto pick2;
        }
        printf("\n");

        /* ProbComplete */
pick3:      printf("Do you wish to set the probabilities of completing a bus access\n");
        printf("(R)andomly\n(D)efault value, or\n(I)ndividually?\n");
        scanf(" %c", &Answer);

        switch ((int)Answer)
        {

            case 'R':
            case 'r':
```

```
                    for(i=0; i<NUMPROCESSORS; i++)
                    {
                        ProcArray[i].ProbComplete = Random();
                        printf("Probability of completion %d:  %f\n",i,ProcArray[i].ProbComplete);
                    }
                    break;
                case 'D':
                case 'd':
                    printf("Default value?\n");
                    scanf("%f", &Value);
                    for (i = 0; i < NUMPROCESSORS; i++)
                        ProcArray[i].ProbComplete = Value;
                    break;
                case 'I':
                case 'i':
                    for (i = 0; i < NUMPROCESSORS; i++)
                    {
                        printf("\nProbability of completion for processor %d:  ",i);
                        scanf("%f", &Value);
                        ProcArray[i].ProbComplete = Value;
                    }
                    printf("\n");
                    break;
                default:
                    printf("Dumb.  Very dumb. Try again.\n");
                    goto pick3;
            }
            printf("\n");
    }   /* End of init() */
```

init Local Symbols

Name	Class	Offset	Register
inregs	auto	-0024	
Answer	auto	-0016	
outregs	auto	-0012	
i	auto	***	si
Value	auto	-0004	

```

    main()
    {
        register int i,j;
        int k;

        init();
        for (i=0; i<MAXLOOPS; i++)
        {
            k = i;
            if (!(i % 1000))
```



```
            {
                printf("%d iterations\n", i);
                for (j=0; j<NUMPROCESSORS; j++)
                    printf("Wait Cycles: %d  \tBlock Cycles:  %d  \tRun cycles:  %d\n",
                        ProcArray[j].TotalWait, ProcArray[j].TotalBlock, ProcArray[j].TotalRun);
                printf("\n\n");
            }
            for (j = 0; j<NUMPROCESSORS; j++)
            {
                if (!ProcArray[j].Active)
                /* Processor is dead.  Do nothing */
                    continue;

                if (j == BusMaster)
                /*  I.  Has bus
                    A:  Dies
                            Mark as dead
                            Release bus
                            Update Priorities
                    B.  Complete access
                            Release bus
                            Update Priorities
                    C.  Continue access
                            Update times
                */
                {
                    if (Random() < ProcArray[j].ProbFail)
                    {
                    /* The processor just failed.  Mark as dead.  Release
                       bus by setting BusMaster to FREE and Priority to
                       FREE.  Update all of the priorities */

                        printf(" Processor %d failed at time %d\n", j, i);
                        ProcArray[j].Active = 0;

                        BusMaster = FREE;
                        Priority = FREE;
                        UpdatePriority();
                        continue;
                    }   /* end of Bus Master failure */

                    /* Check to see if process will complete during
```

```
                 this bus cycle.  Else continue to hold bus. */

              RunProcess(ProcArray[j].ProbComplete);
              continue;

          } /* end of processes if processor already has bus */

          /*  II.  Does not have bus.
                   A. Dies
                      Mark as dead
                      1.  If blocked or waiting
                            Update Total/LongestBlock/Wait
```



```
                            Clear block and end wait
                   B. Does not die
                      1. If blocked, waiting, or makes request
                         a.   If gets bus
                           i.   If bus active
                                Mark bus master as blocked
                                Increment block time
                                Get bus
                                    Update Total/LongestBlock/Wait
                                    Clear Blocked/Waiting flags
                                    Update priority
                                    Update times
                             ii.    If bus inactive
                                    Get bus
                                    Update times
                           b.     If not get bus
                                 If not blocked or waiting,
                                        mark as blocked.
                                 update times
                      2. No request
                            Do nothing.
          */
          if (Random() < ProcArray[j].ProbFail)
          {
          /* The processor just failed.  Mark as dead. */

             printf(" Processor %d failed at time %d\n", j, i);
             ProcArray[j].Active = 0;

             if (ProcArray[j].Blocked || ProcArray[j].Waiting)
             /* If the failed processor was blocked or waiting,
                update the Total/LongestBlock/Wait blocks and clear
                the Blocked/Waiting flags */
             {
                if(ProcArray[j].PresentWait > ProcArray[j].LongestWait)
                     ProcArray[j].LongestWait = ProcArray[j].PresentWait;
                if(ProcArray[j].PresentBlock>ProcArray[j].LongestBlock)
                     ProcArray[j].LongestBlock = ProcArray[j].PresentBlock;
                ProcArray[j].TotalWait += ProcArray[j].PresentWait;
                ProcArray[j].TotalBlock += ProcArray[j].PresentBlock;
                ProcArray[j].PresentWait = 0;
                ProcArray[j].PresentBlock = 0;
                ProcArray[j].Blocked = 0;
                ProcArray[j].Waiting = 0;

             }
             /* else do nothing */
             continue;
          }  /* end of failed processor */

          /*  Else it did not die.  Check and see if it is waiting,
              blocked, or making a bus request. */
          if(ProcArray[j].Waiting||ProcArray[j].Blocked ||
              Random() <= ProcArray[j].ProbRequest)
          {
```



```
          /* Has made request for bus.  See if it will get it */
             if(ProcArray[j].Priority > Priority)
             {
                if (Priority != FREE)
                /* Bus already active.  Block and replace old
                   bus master */
                {
                   ProcArray[BusMaster].Blocked++;
                   ProcArray[BusMaster].NumBlocks++;
                   UpdatePriority();
                   BusMaster = j;
                   ProcArray[j].Running = 1;
                   ProcArray[j].NumRuns++;
                   Priority = ProcArray[j].Priority;

                   if(ProcArray[j].PresentWait > ProcArray[j].LongestWait)
                        ProcArray[j].LongestWait = ProcArray[j].PresentWait;
                   if(ProcArray[j].PresentBlock>ProcArray[j].LongestBlock)
                        ProcArray[j].LongestBlock = ProcArray[j].PresentBlock;
                   ProcArray[j].TotalWait += ProcArray[j].PresentWait;
                   ProcArray[j].TotalBlock += ProcArray[j].PresentBlock;
                   ProcArray[j].PresentWait = 0;
                   ProcArray[j].PresentBlock = 0;
                   ProcArray[j].Blocked = 0;
                   ProcArray[j].Waiting = 0;
                   RunProcess(ProcArray[j].ProbComplete);
                }
```

```c
                else
                /* Bus is free */
                {
                    BusMaster = j;
                    Priority = ProcArray[j].Priority;
                    ProcArray[j].Running = 1;
                    ProcArray[j].NumRuns++;

                    if(ProcArray[j].PresentWait > ProcArray[j].LongestWait)
                        ProcArray[j].LongestWait = ProcArray[j].PresentWait;
                    if(ProcArray[j].PresentBlock>ProcArray[j].LongestBlock)
                        ProcArray[j].LongestBlock = ProcArray[j].PresentBlock;
                    ProcArray[j].TotalWait += ProcArray[j].PresentWait;
                    ProcArray[j].TotalBlock += ProcArray[j].PresentBlock;
                    ProcArray[j].PresentWait = 0;
                    ProcArray[j].PresentBlock = 0;
                    ProcArray[j].Blocked = 0;
                    ProcArray[j].Waiting = 0;
                    RunProcess(ProcArray[j].ProbComplete);
                    continue;
                } /* end else bus is free */
                continue;
            } /* end if get bus */

            /* Else will not get bus.  Put on wait if not
               blocked or waiting already.  Update times.
            */
```



```c
            if (!ProcArray[j].Blocked && !ProcArray[j].Waiting)
            {
                ProcArray[j].Waiting = 1;
                ProcArray[j].NumWaits++;
            }
            continue;
          } /* end if process alive */
       } /* end processor loop */
       UpdateTime();
    } /* End main loop */
    printf("\n");
    for (i = 0; i < NUMPROCESSORS; i++)
    {
        SumWaitTime += (long) ProcArray[i].TotalWait;
        SumBlockTime += ((long) ProcArray[i].TotalBlock;
        SumRunTime += (long) ProcArray[i].TotalRun;
        SumNumWaits += ((long)ProcArray[i].NumWaits;
        SumNumBlocks += ((long)ProcArray[i].NumBlocks;
        SumNumRuns += (long) ProcArray[i].NumRuns;

        if(!ProcArray[i].NumWaits)
            printf("Processor %d never waited!\n",i);

        if(!ProcArray[i].NumBlocks)
            printf("Processor %d was never blocked!\n",i);

        if(!ProcArray[i].NumRuns)
            printf("Processor %d never had the bus!\n", i);

        if(MaxWaitTime < ProcArray[i].LongestWait)
            MaxWaitTime = ProcArray[i].LongestWait;

        if(MaxBlockTime < ProcArray[i].LongestBlock)
            MaxBlockTime = ProcArray[i].LongestBlock;

        if(MaxRunTime < ProcArray[i].LongestRun)
            MaxRunTime = ProcArray[i].LongestRun;

        if(MaxNumWaits < ProcArray[i].NumWaits)
            MaxNumWaits = ProcArray[i].NumWaits;

        if(MaxNumBlocks < ProcArray[i].NumBlocks)
            MaxNumBlocks = ProcArray[i].NumBlocks;

        if(MaxNumRuns < ProcArray[i].NumRuns)
            MaxNumRuns = ProcArray[i].NumRuns;
    }
    printf("\nTotal number of cycles blocked:  %ld\n", SumBlockTime);
    printf("Total number of cycles in wait:  %ld\n", SumWaitTime);
    printf("Total number of cycles on bus:   %ld\n", SumRunTime);
    printf("Longest time for a block:   %d\n", MaxBlockTime);
    printf("Longest time for a wait:    %d\n", MaxWaitTime);
    printf("Longest time in run:        %d\n", MaxRunTime);
```



```c
    printf("Average time in a block:   %ld\n", SumBlockTime/SumNumBlocks);
    printf("Average time in a wait:    %ld\n", SumWaitTime/SumNumWaits);
    printf("Average time running:      %ld\n", SumRunTime/SumNumRuns);

    MaxNumBlocks ?
        printf("Max. number of blocks for a processor:  %d\n", MaxNumBlocks)
        : printf("Nobody was ever blocked!!\n");

    MaxNumWaits ?
        printf("Max. number of waits for a processor:   %d\n", MaxNumWaits)
        : printf("Nobody ever had to wait!!!\n");
```

```
        MaxNumRuns ?
            printf("Max. number of runs for a processor:    %d\n", MaxNumRuns)
            : printf("Nobody ever had the bus!!\n");

 } /* End main */
```

main Local Symbols

Name	Class	Offset	Register
k	auto	-0006	
i	auto	***	di
j	auto	***	si

Global Symbols

Name	Type	Size	Class	Offset
AvgBlock	struct/array	20	common	***
AvgRun	struct/array	20	common	***
AvgWait	struct/array	20	common	***
BusMaster	int	2	global	0000
MaxBlockTime	int	2	global	001e
MaxNumBlocks	int	2	global	0024
MaxNumRuns	int	2	global	0026
MaxNumWaits	int	2	global	0022
MaxRunTime	int	2	global	0020
MaxWaitTime	int	2	global	001c
Priority	int	2	global	0002
ProcArray	struct/array	420	common	***
Random	near function	***	global	0000
RunProcess	near function	***	global	017a
Seed	long	4	common	***
SetPriority	near function	***	global	00f0
SumAvgBlock	int	2	global	002a
SumAvgRun	int	2	global	002c
SumAvgWait	int	2	global	0028
SumBlockTime	long	4	global	0008

Global Symbols

Name	Type	Size	Class	Offset
SumNumBlocks	long	4	global	0014
SumNumRuns	long	4	global	0018
SumNumWaits	long	4	global	0010
SumRunTime	long	4	global	000c
SumWaitTime	long	4	global	0004
UpdatePriority	near function	***	global	012e
UpdateTime	near function	***	global	0042
init	near function	***	global	01b6
intdos	near function	***	extern	***
main	near function	***	global	051a
printf	near function	***	extern	***
scanf	near function	***	extern	***

```
Code size = 0a18 (2584)
Data size = 0617 (1559)
Bss size  = 0000 (0)
```

No errors detected

APPENDIX B

RESULTS OF RUNS

A. Run 1: Random Loading

```
Do you wish to set the probabilities of failure
(R)andomly
(D)efault value, or
(I)ndividually?
R
Probability of failure 0:  0.000067
Probability of failure 1:  [illegible]
Probability of failure 2:  [illegible]
Probability of failure 3:  [illegible]
Probability of failure 4:  [illegible]
Probability of failure 5:  [illegible]
Probability of failure 6:  [illegible]
Probability of failure 7:  [illegible]
Probability of failure 8:  [illegible]
Probability of failure 9:  0.000040
Do you wish to set the probabilities of making a bus request
(R)andomly
(D)efault value, or
(I)ndividually?
R
Probability of Bus Request 0:  0.463150
Probability of Bus Request 1:  [illegible]
Probability of Bus Request 2:  [illegible]
Probability of Bus Request 3:  [illegible]
Probability of Bus Request 4:  [illegible]
Probability of Bus Request 5:  [illegible]
Probability of Bus Request 6:  [illegible]
Probability of Bus Request 7:  [illegible]
Probability of Bus Request 8:  [illegible]
Probability of Bus Request 9:  0.761528
Do you wish to set the probabilities of completing a bus access
(R)andomly
(D)efault value, or
(I)ndividually?
R
Probability of completion 0:  0.116318
Probability of completion 1:  [illegible]
Probability of completion 2:  [illegible]
Probability of completion 3:  [illegible]
Probability of completion 4:  [illegible]
Probability of completion 5:  [illegible]
Probability of completion 6:  [illegible]
Probability of completion 7:  [illegible]
Probability of completion 8:  [illegible]
Probability of completion 9:  0.335708
0 iterations
Wait Cycles: 0     Block Cycles: 0     Run cycles: 0
Wait Cycles: 0     Block Cycles: 0     Run cycles: 0
Wait Cycles: 0     Block Cycles: 0     Run cycles: 0
Wait Cycles: 0     Block Cycles: 0     Run cycles: 0
Wait Cycles: 0     Block Cycles: 0     Run cycles: 0
Wait Cycles: 0     Block Cycles: 0     Run cycles: 0
Wait Cycles: 0     Block Cycles: 0     Run cycles: 0
Wait Cycles: 0     Block Cycles: 0     Run cycles: 0
Wait Cycles: 0     Block Cycles: 0     Run cycles: 0
Wait Cycles: 0     Block Cycles: 0     Run cycles: 0

 Processor 7 failed at time 692
1000 iterations
Wait Cycles: 108          Block Cycles: 212          Run cycles: 638
Wait Cycles: [illegible]  Block Cycles: [illegible]  Run cycles: [illegible]
Wait Cycles: [illegible]  Block Cycles: [illegible]  Run cycles: 101
Wait Cycles: [illegible]  Block Cycles: 137          Run cycles: 103
Wait Cycles: [illegible]  Block Cycles: 121          Run cycles: [illegible]
Wait Cycles: [illegible]  Block Cycles: 77           Run cycles: [illegible]
Wait Cycles: [illegible]  Block Cycles: 769          Run cycles: [illegible]
Wait Cycles: [illegible]  Block Cycles: 379          Run cycles: [illegible]
Wait Cycles: [illegible]  Block Cycles: 377          Run cycles: [illegible]
Wait Cycles: [illegible]  Block Cycles: 0            Run cycles: [illegible]
Wait Cycles: 855          Block Cycles: 207          Run cycles: 138

 Processor 6 failed at time 1636
 Processor 0 failed at time 1485
2000 iterations
Wait Cycles: 297          Block Cycles: 280          Run cycles: 836
Wait Cycles: 700          Block Cycles: 616          Run cycles: 172
Wait Cycles: [illegible]  Block Cycles: 217          Run cycles: 202
Wait Cycles: [illegible]  Block Cycles: [illegible]  Run cycles: [illegible]
Wait Cycles: [illegible]  Block Cycles: [illegible]  Run cycles: 240
Wait Cycles: [illegible]  Block Cycles: 1130         Run cycles: 1014
Wait Cycles: [illegible]  Block Cycles: [illegible]  Run cycles: [illegible]
Wait Cycles: [illegible]  Block Cycles: 377          Run cycles: 60
Wait Cycles: [illegible]  Block Cycles: 0            Run cycles: [illegible]
Wait Cycles: 1181         Block Cycles: 509          Run cycles: 300

3000 iterations
Wait Cycles: 297          Block Cycles: 280          Run cycles: 836
Wait Cycles: [illegible]  Block Cycles: 701          Run cycles: [illegible]
Wait Cycles: [illegible]  Block Cycles: [illegible]  Run cycles: [illegible]
Wait Cycles: [illegible]  Block Cycles: [illegible]  Run cycles: [illegible]
Wait Cycles: [illegible]  Block Cycles: [illegible]  Run cycles: [illegible]
Wait Cycles: [illegible]  Block Cycles: 1469         Run cycles: 1622
Wait Cycles: [illegible]  Block Cycles: [illegible]  Run cycles: [illegible]
Wait Cycles: [illegible]  Block Cycles: 377          Run cycles: 60
Wait Cycles: [illegible]  Block Cycles: 0            Run cycles: 179
Wait Cycles: 1822         Block Cycles: 590          Run cycles: 561
```

```
4000 Iterations
Wait cycles: 277        Block Cycles: 280       Run cycles: 834
[illegible]
Wait cycles: 2283       Block Cycles: 899       Run cycles: 712

 Processor 5 failed at time 4633
5000 Iterations
Wait cycles: 277        Block Cycles: 280       Run cycles: 834
[illegible]
Wait cycles: 2933       Block Cycles: 929       Run cycles: 1062

 Processor 4 failed at time 5854
6000 Iterations
Wait cycles: 277        Block Cycles: 280       Run cycles: 834
[illegible]
Wait cycles: 3086       Block Cycles: 964       Run cycles: 1798

 Processor 3 failed at time 6817
7000 Iterations
Wait cycles: 277        Block Cycles: 280       Run cycles: 834
[illegible]
Wait cycles: 3163       Block Cycles: 975       Run cycles: 2369

 Processor 9 failed at time 7906
8000 Iterations
Wait cycles: 277        Block Cycles: 280       Run cycles: 834
[illegible]
Wait cycles: 3243       Block Cycles: 985       Run cycles: 3285

9000 Iterations
Wait cycles: 277        Block Cycles: 280       Run cycles: 834
[illegible]
Wait cycles: 3243       Block Cycles: 985       Run cycles: 3285

Processor 8 was never blocked!

Total number of cycles blocked: 7654
Total number of cycles in wait: [illegible]
Total number of cycles on bus: 15229
Longest time for a block: [illegible]
Longest time for a wait: [illegible]
Longest time in run: [illegible]
Average time in a block: [illegible]
Average time in a wait: [illegible]
Average time running: 1
Max. number of blocks for a processor: 618
Max. number of waits for a processor: [illegible]
Max. number of runs for a processor: 2269
```

B. Run 2: Random Loading

```
Do you wish to set the probabilities of failure
(R)andomly
(D)efault value, or
(I)ndividually?
R
Probability of failure 0: 0.000063
Probability of failure 1: 0.000048
```

```
Probability of failure 2:  0.000016
Probability of failure 3:  0.000050
Probability of failure 4:  0.000053
Probability of failure 5:  0.000047
Probability of failure 6:  0.000073
Probability of failure 7:  0.000057
Probability of failure 8:  0.000020
Probability of failure 9:  0.000046

Do you wish to set the probabilities of making a bus request
(R)andomly
(D)efault value, or
(I)ndividually?
R
Probability of Bus Request 0:  0.219482
Probability of Bus Request 1:  0.252525
Probability of Bus Request 2:  0.254242
Probability of Bus Request 3:  0.253759
Probability of Bus Request 4:  0.320535
Probability of Bus Request 5:  0.950516
Probability of Bus Request 6:  0.577220
Probability of Bus Request 7:  0.860631
Probability of Bus Request 8:  0.139262
Probability of Bus Request 9:  0.362808

Do you wish to set the probabilities of completing a bus access
(R)andomly
(D)efault value, or
(I)ndividually?
R
Probability of completion 0:  0.182831
Probability of completion 1:  0.611313
Probability of completion 2:  0.790100
Probability of completion 3:  0.401077
Probability of completion 4:  0.329229
Probability of completion 5:  0.052274
Probability of completion 6:  0.797263
Probability of completion 7:  0.237198
Probability of completion 8:  0.195451
Probability of completion 9:  0.951523


Total number of cycles blocked:  26012
Total number of cycles in wait:  32804
Total number of cycles on bus:   16821
Longest time for a block:   150
Longest time for a wait:    148
Longest time in run:        140
Average time in a block:    6
Average time in a wait:     13
Average time running:       2
Max. number of blocks for a processor:  904
Max. number of waits for a processor:   432
Max. number of runs for a processor:    1211
```

C. Run 3: Random Loading

```
Do you wish to set the probabilities of failure
(R)andomly
(D)efault value, or
(I)ndividually?
R
Probability of failure 0:  0.000073
Probability of failure 1:  0.000060
Probability of failure 2:  0.000038
Probability of failure 3:  0.000021
Probability of failure 4:  0.000026
Probability of failure 5:  0.000029
Probability of failure 6:  0.000068
Probability of failure 7:  0.000026
Probability of failure 8:  0.000053
Probability of failure 9:  0.000022

Do you wish to set the probabilities of making a bus request
(R)andomly
(D)efault value, or
(I)ndividually?
R
Probability of Bus Request 0:  0.003633
Probability of Bus Request 1:  0.635926
Probability of Bus Request 2:  0.126838
Probability of Bus Request 3:  0.618625
Probability of Bus Request 4:  0.421702
Probability of Bus Request 5:  0.472274
Probability of Bus Request 6:  0.348276
Probability of Bus Request 7:  0.375698
Probability of Bus Request 8:  0.377825
Probability of Bus Request 9:  0.146576

Do you wish to set the probabilities of completing a bus access
(R)andomly
(D)efault value, or
(I)ndividually?
R
Probability of completion 0:  0.967168
Probability of completion 1:  0.223572
Probability of completion 2:  0.183420
Probability of completion 3:  0.737635
Probability of completion 4:  0.150536
Probability of completion 5:  0.144861
Probability of completion 6:  0.881119
Probability of completion 7:  0.914227
Probability of completion 8:  0.158014
Probability of completion 9:  0.379700
```

```
Processor 0 was never blocked!
Processor 6 was never blocked!

Total number of cycles blocked:  16211
Total number of cycles in wait:  18857
Total number of cycles on bus:   16605
Longest time for a block:  79
Longest time for a wait:   70
Longest time in run:       56
Average time in a block:   4
Average time in a wait:    7
Average time running:      2
Max. number of blocks for a processor:  1259
Max. number of waits for a processor:   722
Max. number of runs for a processor:    1778
```

D. Run 4: High Probability of Request/Low Probability of Completion

```
Do you wish to set the probabilities of failure
(R)andomly
(D)efault value, or
(I)ndividually?
D
Default value?
.00001

Do you wish to set the probabilities of making a bus request
(R)andomly
(D)efault value, or
(I)ndividually?
D
Default value?
.8

Do you wish to set the probabilities of completing a bus access
(R)andomly
(D)efault value, or
(I)ndividually?
D
Default value?
.2


Total number of cycles blocked:  56845
Total number of cycles in wait:  27538
Total number of cycles on bus:   25441
Longest time for a block:  108
Longest time for a wait:   96
Longest time in run:       50
Average time in a block:   4
Average time in a wait:    7
Average time running:      1
Max. number of blocks for a processor:  1692
Max. number of waits for a processor:   599
Max. number of runs for a processor:    2570
```

E. Run 5: High Probability of Request/High Probability of Completion

```
Do you wish to set the probabilities of failure
(R)andomly
(D)efault value, or
(I)ndividually?
D
Default value?
.00001

Do you wish to set the probabilities of making a bus request
(R)andomly
(D)efault value, or
(I)ndividually?
D
Default value?
.8

Do you wish to set the probabilities of completing a bus access
(R)andomly
(D)efault value, or
(I)ndividually?
D
Default value?
.8


Total number of cycles blocked:  11000
Total number of cycles in wait:  18528
Total number of cycles on bus:   58000
Longest time for a block:  8
Longest time for a wait:   7
Longest time in run:       4
Average time in a block:   1
Average time in a wait:    1
Average time running:      1
Max. number of blocks for a processor:  1138
Max. number of waits for a processor:   1562
Max. number of runs for a processor:    6270
```

F. Run 6: Low Probability of Request/Low Probability of Completion

```
Do you wish to set the probabilities of failure
(R)andomly
(D)efault value, or
(I)ndividually?
D
Default value?
.00001

Do you wish to set the probabilities of making a bus request
(R)andomly
(D)efault value, or
(I)ndividually?
D
Default value?
.2

Do you wish to set the probabilities of completing a bus access
(R)andomly
(D)efault value, or
(I)ndividually?
D
Default value?
.2


Total number of cycles blocked:  42550
Total number of cycles in wait:  21397
Total number of cycles on bus:   19677
Longest time for a block:  36
Longest time for a wait:   65
Longest time in run:       48
Average time in a block:   5
Average time in a wait:    8
Average time running:      1
Max. number of blocks for a processor:  1053
Max. number of waits for a processor:   384
Max. number of runs for a processor:    1525
```

G. Run 7: Low Probability of Request/High Probability of Completion

```
Do you wish to set the probabilities of failure
(R)andomly
(D)efault value, or
(I)ndividually?
D
Default value?
.00001

Do you wish to set the probabilities of making a bus request
(R)andomly
(D)efault value, or
(I)ndividually?
D
Default value?
.2

Do you wish to set the probabilities of completing a bus access
(R)andomly
(D)efault value, or
(I)ndividually?
D
Default value?
.8


Total number of cycles blocked:  1968
Total number of cycles in wait:  5265
Total number of cycles on bus:   22729
Longest time for a block:  5
Longest time for a wait:   6
Longest time in run:       6
Average time in a block:   0
Average time in a wait:    1
Average time running:      1
Max. number of blocks for a processor:  286
Max. number of waits for a processor:   412
Max. number of runs for a processor:    2119
```

DISTRIBUTION

	No. Copies
US Army Materiel System Analysis Activity ATTN: AMXSY-MP Aberdeen Proving Ground, MD 21005	1
IIT Research Institute ATTN: GACIAC 10 W. 35th Street Chicago, IL 60616	1
AMSMI-RD, Dr. McCorkle	1
Dr. Rhoades	1
-RD-AS-SS, Mr. Sims	1
-RD-BA	1
-RD-GC, Mr. Sliz	1
-RD-GC-S	5
-RD-CS-R	15
-RD-CS-T	1
-GC-IP	1
AMCPM-ML, Mr. Yarbrough	1
-ML-TM, Mr. Crosswhite	1
-ML-TM-R, Mr. Hill	1